Learn How to Encode Categorical Data with Pandas factorize()

Authored by

Mohammed looti

November 1, 2025

Introduction to Categorical Encoding with factorize()

The transformation of qualitative data into a quantifiable format is a critical, prerequisite step in nearly every data science workflow. To facilitate this fundamental requirement, the powerful **pandas** library offers an indispensable tool: the `factorize()` function. This function provides a robust and highly efficient mechanism specifically designed to **encode strings**, or any other categorical labels, into streamlined **numeric values**.

Data professionals routinely utilize this technique, generally referred to as categorical encoding, when preparing datasets for consumption by machine learning models. Since most models inherently require input features to be represented numerically, converting textual categories is essential for successful training. The `factorize()` method systematically assigns a unique integer code to every distinct categorical value found within a specified **DataFrame** column, thereby guaranteeing data consistency while minimizing memory overhead.

This comprehensive guide is dedicated to exploring the effective application of the **factorize()** function across various operational scopes within a **pandas DataFrame**. We will move from the simplest case--encoding a single, targeted column--to advanced strategies involving mass transformation of the entire dataset, ensuring readers grasp the versatility and performance benefits of this encoding approach.

Decoding the factorize() Output: Codes and Uniques

Before implementing any encoding strategy, it is paramount to understand precisely what the **factorize()** function returns. Unlike simpler encoding methods that might return only a single data series, `factorize()` returns a **tuple** composed of two distinct elements: the array of **codes** and the array of **unique values**. Recognizing the purpose of each component is key to using the function correctly for data transformation.

The first element of the returned tuple, which is the array of **codes**, represents the zero-based integer mapping of the original input data. These integers are generated based on the order in which the unique values first appear in the input column. Crucially, these codes maintain a direct correspondence to the original input strings, preserving the sequence of data points. For example, if the first time 'North' appears, it is encoded as 0, every subsequent occurrence of 'North' will also be 0.

The second element is the array of **unique values**, often termed the "uniques" or "labels." This array acts as the lookup table, mapping the generated integer codes back to their original categorical representations. If 'West' was encoded as 0, 'West' will occupy the index 0 position in the array of unique values. This array is invaluable for inverse transformations or for generating documentation of the encoding scheme used.

For the primary purpose of preparing data for numerical models, we are typically interested solely in the new numerical representation. Consequently, when applying `factorize()` to a column within a **DataFrame**, we nearly always select the first element of the returned tuple using the index . This targeted selection allows us to efficiently overwrite the original, textual string data with its newly generated, zero-based integer codes.

Strategic Approaches for Applying factorize()

The inherent flexibility of the **pandas** library enables several distinct approaches for applying `factorize()`, depending entirely on the breadth of the encoding required. Whether the task involves a single, isolated column or a wide range of categorical features, there is an optimal method to ensure efficiency and code clarity.

We will analyze three primary methods that showcase increasing levels of operational scope. These techniques range from highly explicit, targeted column assignment to broad, vectorized transformation using the powerful `apply()` method, which is the standard mechanism for high-performance, column-wise operations in pandas.

Method 1: Factorize One Column: A direct, explicit assignment used for targeted encoding of a single variable.
Method 2: Factorize Specific Columns: Encoding a designated subset of columns simultaneously by leveraging the `apply()` method on a sliced **DataFrame**.
Method 3: Factorize All Columns: Applying the transformation across every column in the **DataFrame**, suitable when the entire dataset is composed of categorical or acceptable data types.

To demonstrate these three methods practically, we will initialize a sample **DataFrame** containing fictional data for a sports team roster. This dataset includes three columns that are stored as string (object) data types, making them perfect candidates for numerical encoding. The following code snippet generates the base **DataFrame** that will be used across all subsequent examples. Note that in a production environment, it is often advisable to create a copy of the **DataFrame** before performing destructive transformations like overwriting string columns.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'conf': ,
'team': ,
'position': })

#view DataFrame
df

conf team position
0 West A Guard
1 West B Forward
2 East C Guard
3 East D Center

Method 1: Targeted Encoding of a Single Column

The simplest and most explicit use case for **`factorize()`** occurs when only one column within the dataset requires numerical transformation. This approach is highly readable and is best suited for targeted data cleaning tasks where the intention is to modify only a single feature. This method involves directly selecting the column, applying the function, and reassigning the results.

To execute this operation, we first select the column of interest--in this case, `'conf'`. We then apply `pd.factorize()` directly to that column's series. Crucially, we immediately access the resulting code array by appending to the function call. This array of integers is then assigned back to the original column, replacing the categorical strings with their new integer codes.

For our example, the `'conf'` column is encoded based on the order of first appearance. Since 'West' is the first unique value encountered, it is assigned the code 0. 'East' is the second unique value, resulting in it receiving the code 1. This straightforward conversion is often sufficient for basic **categorical encoding** tasks.

#factorize the conf column only
df = pd.factorize(df)

#view updated DataFrame
df

conf team position
0 0 A Guard
1 0 B Forward
2 1 C Guard
3 1 D Center

Upon reviewing the updated **DataFrame**, we confirm that only the `'conf'` column has been transformed. Every original entry of 'West' now maps to 0, and 'East' maps to 1. The remaining columns, `'team'` and `'position'`, retain their original string format, demonstrating the surgical precision of this single-column assignment method.

Method 2: Efficiently Encoding Specific Columns using apply()

When preparing a dataset for advanced processing, it is common to require the encoding of several columns while leaving others (such as numerical features) unmodified. In these situations, employing the `apply()` method on a subset of the **DataFrame** provides the most scalable and efficient solution, significantly reducing repetitive code and harnessing **pandas'** internal vectorization capabilities.

This approach begins by selecting the target columns using a list of column names, effectively slicing the **DataFrame** (e.g., `df]`). We then invoke the `apply()` method on this subset, supplying a lambda function that iteratively executes `pd.factorize(x)` on each column (represented by `x`). The results are then assigned back to the original slice, transforming multiple columns in one clean step.

This technique is particularly valuable in the context of large-scale data preparation where datasets contain mixed data types. It allows data scientists to selectively factorize only the non-numeric, categorical features while ensuring that existing numerical columns (like continuous measurements or counts) remain untouched, preventing the loss of crucial quantitative information that could distort **machine learning models**.

#factorize conf and team columns only
df] = df].apply(lambda x: pd.factorize(x))

#view updated DataFrame
df

conf team position
0 0 0 Guard
1 0 1 Forward
2 1 2 Guard
3 1 3 Center

The resulting output confirms the transformation of both the `'conf'` and `'team'` columns. The team names ('A', 'B', 'C', 'D') have been sequentially encoded as 0, 1, 2, and 3, based on their appearance order. Crucially, the `'position'` column remains unchanged, illustrating the precision of the targeted `apply()` method.

Method 3: Mass Transformation of the Entire DataFrame

In scenarios where a **DataFrame** is composed entirely of categorical columns that necessitate numerical representation, the broadest application of the technique involves applying the

`factorize()` operation across all columns simultaneously. This provides the most concise syntax for a complete dataset transformation and is frequently utilized when working with smaller, purely qualitative datasets.

We achieve this comprehensive encoding by calling `df.apply()` without explicitly specifying a subset of columns. In this configuration, the provided lambda function is automatically executed column-wise across the entire **DataFrame**, ensuring every feature is encoded based on its unique values.

However, a significant caution must be observed when using this method: it requires absolute assurance that all columns are suitable for factorization. Applying `factorize()` to columns that are already numerical (such as floating-point monetary values or large integer IDs) will likely lead to unintended consequences. Since `factorize()` treats every unique value as a distinct category, pre-existing numerical columns may be encoded in a way that destroys their continuous nature or results in unexpected index mapping rather than preserving their intrinsic quantitative meaning.

#factorize all columns
df = df.apply(lambda x: pd.factorize(x))

#view updated DataFrame
df

conf team position
0 0 0 0
1 0 1 1
2 1 2 0
3 1 3 2

The final output confirms that every feature--`'conf'`, `'team'`, and `'position'`--has been successfully converted into its corresponding numerical code. For the `'position'` column, the strings were mapped such that 'Guard' is 0, 'Forward' is 1, and 'Center' is 2. This process successfully completes the full categorical encoding of the sample dataset using the broadest application of `factorize()`.

Beyond factorize(): Next Steps in Data Preparation

While `factorize()` is an outstanding utility for straightforward label encoding, more sophisticated data science tasks may demand alternative encoding strategies. Scenarios requiring techniques like one-hot encoding (for nominal variables) or specific ordinal encoding (where the order of categories matters) often necessitate using different tools available either within **pandas** or specialized libraries like scikit-learn.

The primary drawback of simple label encoding via **`factorize()`** is the implicit introduction of an ordinal relationship between categories. For instance, if 'Red' is 0 and 'Blue' is 1, a machine learning algorithm might mistakenly infer that 'Blue' is "greater" than 'Red'. For categories where no such order exists, methods like one-hot encoding are preferred to avoid this bias.

For data professionals seeking to deepen their expertise in data preparation and transformation techniques, the following resources are recommended for expanding knowledge beyond simple label encoding and mastering advanced **feature engineering** methods in Python.

The official **pandas** documentation on `factorize()`, which covers advanced parameters like handling missing values.
Detailed tutorials on using `pd.get_dummies()`, the standard **pandas** function for implementing one-hot encoding.
Guides on advanced **categorical encoding** techniques using scikit-learn's preprocessing modules, such as `OrdinalEncoder` or `OneHotEncoder`.